形式的検証機能を備えたインターネットエージェントプログラミングシステム

櫟　粛之
日本電信電話株式会社 NTT コミュニケーション科学基礎研究所

あらまし　本論文では，形式的検証機能を備えたエージェントプログラミングシステムを提案する．その言語は，if-then ルールに基づく単純なセマンティクスをもち，またインターネットエージェントの動作を記述するには十分な，いくつかのエージェントのアクションが与えられている．形式的検証に関しては，CTL のモデルチェックアルゴリズムを利用する．その入力として，エージェントシステムのふるまいの論理的記述が必要であり，従来はこの記述を人手により作成していたが，ここで導入される言語では，それを自動的に合成することが可能になる．また，検証すべき要求仕様に対して，その仕様に関連するプログラムの部分だけを自動的に抽出する方法を導入し，自動合成される状態遷移の論理的記述を簡約化することも可能にした．この言語に対する処理系は，Java 上の interpreter として実装されており，Jade などの他のエージェント開発環境にひけをとらない実行スピードを実現している．

An Internet Agent System with a Formal Verification Facility

Tadashi Araragi
[1] NIPPON TELEGRAPH AND TELEPHONE CORPORATION, NTT Communication Science Laboratories

Abstract: In this paper, we present an Internet agent system with a formal verification facility for the agent programs. The language has a simple semantics of "if-then rule" and a few selected actions of agents which are enough to describe agents' behavior working in the Internet. For the verification, we employ the model checking algorithm of CTL as the base engine. The language is designed so that a logical representation of behavior of agents, an input of a CTL model checker, which was manually created by programmers, is automatically derived from the programs. We also introduce a method of reducing the logical representation of behavior by extracting a part of programs which is essentially related to a given requirement specification to be verified. This system is implemented on Java and shows practical speed of execution compared with other agent systems such as Jade..

1. まえがき

昨今インターネットの利用は急速に拡大し，その利用形態も，これまでのメールやWWWなどの情報交換から，Webサービスのようにネットワーク上でシステムが連携してサービスを提供する枠組みに発展しつつある．このWebサービスの枠組みをより細かい個人レベルに推し進めたものがインターネットエージェントである．そこではエージェントとよばれるソフトウェアが人間に代わってインターネット上で，情報の収集や，電子商取引の契約などを行う．各々のエージェントはそれぞれの目的をもって独立に開発され，ネットワーク上でメッセージの送受信を通して他のエージェントと協調し，その目的を達成する．

インターネットエージェントの普及における大きな課題の一つは，簡単に，間違いのないエージェントを作る開発環境の提供である．エージェントプログラミングは非同期分散プログラミングでもある．非同期分散プログラムでは通信（メッセージ送受信）のタイミングによって無数の可能な実行（列）が存在し，熟練したプログラマでも，自分のプログラムが全ての可能な実行で正しく動作するかを検証することは難しい．

上記の問題に対する一つのアプローチとして形式的検証手法の利用が考えられる．形式的検証では，システムの抽象的な実行により，与えられた要求仕様が全ての場合(可能な実行)において満足されるかを論理的な形で効率良くチェックされる．しかし，これらの検証手法を適用するには，システムの動作を一旦論理的な形で表現し直す必要があり，それを間違いなく行うにはやはり相当の熟練が必要となる．また，形式的検証のかかえる大きな問題の一つとして，探索空間の爆発があり，対象とするシステムをあらかじめ適切に抽象化することも要求される．

これらの問題を解決するため，本論文では，直接形式的検証可能なエージェントプログラム言

語のデザイン(Erdoes)と,探索空間の爆発を押さえるためのシステム状態遷移モデルの簡約化の手法を提案する．提案する言語は，if thenルールにもとづく単純なセマンティクスを持ち,そのセマンティクスに基づいて時相論理(F)CTLのシンボリックモデルチェックに必要なシステム動作の論理式をプログラムから自動合成する.状態遷移モデルの簡約化では,検証する要求仕様が与えられたとき,その仕様に影響を及ぼしうるプログラムの部分のみを自動で抽出する抽象化の手法を導入する.

2．インターネットエージェントシステムの計算モデルとそのプログラミング言語

2.1 インターネットエージェントの計算モデル

まず,分散システムの観点から,インターネットエージェントの計算モデルを定める[1].

プロセスとしての各エージェントはグローバルに一意な名前を持つ.エージェントが複数いる場合,その全体の動作はfairなinterleavingモデルに従う.即ち,実行の各ステップにおいてアトミックな動作を行うエージェントは一つだけであり,各エージェントはどの時点から見ても,いつかは動作の順番が与えられる.

エージェントはメッセージベースと呼ばれるメッセージの集合を持つ.ここには,他のエージェントから受信したメッセージと，localな処理の結果をメッセージの形にしたものを保持する.一般の分散システムのモデル(例えばSDL)では,このストーレッジはqueueになるが，メッセージ遅延の幅が大きいインターネットでのアプリケーションを考えた場合,これは集合で十分である.

エージェントは他のエージェントに,その受信エージェントの名前と,送信するメッセージを指定して,メッセージを送信することができる.通信リンクはFIFO queueで，メッセージはハンドシェイクなしにメッセージベースに受け取られる.この規定は,エージェントがあらかじめどのようなエージェントからメッセージが来るかを知らないこと,およびエージェントの自律的な側面から,特定のメッセージの到着を待たずに,それまでにとどいたメッセージをもとに行動を選択する挙動も許すためである.

エージェントはプログラムあるいは状態遷移関係を持ち,メッセージベースの状態（集合としての内容）に基づき,メッセージの送信，あるいはメッセージの書き換えなどの現在のアトミックな動作を決定する.

2.2 エージェントプログラミング言語（Erdoes）

上記の計算モデルを反映した直接形式的検証可能なプログラム言語（Erdoesと呼ぶ）を以下に導入する.

[1]プログラムの構造

まず,プログラムは複数のサブプログラムからなり，各サブプログラムはif-thenルールの列からなる.サブプログラムの一つは実行のエントリーを示す*main*をサブプログラム名として持つ.サブプログラム実行時には原則として先頭のif-thenルールから末尾に向かって順に処理を進める.サブプログラムから別のサブプログラムへの実行の遷移は,下記に説明する*call*アクションにより行われる.

[2]if-thenルール

if-thenルールのif部には，一階述語論理式で,アトム式を*and*または*or*の論理結合子で結んだ式が指定される（メッセージは，一階述語論理式のグランドアトム式としてあらわされる）．そしてthen以下,およびelse以下には,それぞれ複数のアクションが並べられる．アクションの中には,if部の論理式の変数を含むものもある.

if-thenルールの実行では，まずそのブール式が現在のメッセージベースの内容から導かれるかをチェックする.ここでif部の式の変数は存在限定されているものとする．例えばif部がif *msg(f(?x))*（*?x*は変数)で，現在のメッセージベースに*msg(f(g(a, b)))*というメッセージがあれば，if部は導出されたことになり，*?x*にはg(a, b)がバインディングされる.もしif部の導出に成功したらthen部分のactionを順に実行し,そうでなければelse部分のactionを順に実行する.action部分の変数に関しては,if部の導出が成功したとき，その式のinstantiationを，action部の変数にも適用して，actionを具体化する.

[3]action

Erdoesで利用されるactionは以下のとおりである.

<メッセージベースを操作するアクション>

- *add(agt_name: φ)*

メッセージφを送信先のエージェント*agt_name*のメッセージベースに加える.このアクションがErdoesで,エージェント間メッセージ通信を行う唯一のアクションである.

- *rm(φ)*

自分のメッセージベースから,もしメッセージφがあればそれを取り除く.

<実行をコントロールするアクション>

- *call(sub_prg_name)*

サブプログラム*sub_prg_name*を呼ぶ.

- *idle*

何もせず，次の実行時も現在実行している同じif-thenルールを実行する.

<プロセスコントロールのアクション>
- *create(new_agt_name, sub_prg_name, arg1,..)*
名前*new_agt_name*をもつ新しいエージェントを生成する．このエージェントのプログラムは*sub_prg_name*を*main*とし，その他のサブプログラムはそこから直接，あるいは間接的によばれるものである．また，メッセージベースの初期状態はメッセージ*arg1,...* からなる．
- *stop(agt_name)*, *resume(agt_name)*, *kill(agt_name)*
これらのアクションはエージェントの動作の停止，再開，エージェントの消滅を行う．
<その他のアクション>
- *go(url)*
*url*で指定されたホストに移動する．
- *ex_call(method_name, arg1,...)*
外部メソッドmethod_nameを引数*arg1,...*で呼ぶ．結果は，メッセージの形でメッセージベースに返される．

3．エージェントプログラムの検証

Erdoesでは，形式的検証の1つである時相論理FCTL(Fair Computational Tree Logic)のシンボリックモデルチェックアルゴリズム[2]を利用する．このモデルチェックアルゴリズムを利用するにはシステムの全動作(状態遷移)を論理的に表現した式が必要になる．

この論理式は，現在の状態を表す変数たち$\{x\}$と，次のステップでの状態を表す変数たち$\{x^\wedge\}$との関係を記述したブール式$R(\{x\}\{x^\wedge\})$である．Erdoesでは，エージェントプログラムが与えられたときこのブール式を自動で導出する．

3.1エージェントプログラムの全遷移関係式への基本変換

こまかな導出方法は割愛して，変換の大枠を説明する．また，説明を単純にするためプログラムには変数が使われていないものとし，さらにプロセスコントロールのactionもないものとする．

[1]テスト式の論理式への変換
エージェントagtのif-thenルールのテスト式φはメッセージをアトム式とするブール式である．このテスト式の各メッセージ*msg*に対して記号*[msg]*を導入する．そして各メッセージに対し，その記号の冠頭にエージェント名を付与した状態変数，即ち*agt.[msg]*を導入する．*agt.[msg]*は，*agt*のメッセージベースにメッセージ*msg*があることを意味している．そしてブール式の構造はそのまま保って，テスト式を表現するブール式を作る．

例えばエージェント*agt*のテスト式φが *(msg1 and msg2) or msg3*の場合，$(agt.[msg1]\wedge agt.[msg2])\vee agt.[msg3]$となる．このブール式を$bf(agt, \varphi)$と書くことにする．

[2]プログラムスタックの状態遷移関係
プログラムの実行遷移を表すため，if-then ルールの単位でのプログラムスタックを導入する．即ち，現在，サブプログラム*sp*の*n*番目のif-thenルールを実行中であれば，スタックのtopの要素は，対$<sp, n>$で，その直後に，次のif-then ルールに実行が移った場合は，その要素の*n*が*n+1*に変わり，別のサブプログラム*sp'*を呼んだ場合には，$<sp', 1>$がスタックに積まれる．

プログラムの*call*関係を追うことで，可能なスタックを全て求め，[1]で得られる論理式を用いて，スタックの遷移関係を論理式に表現しなおす．ここで，求められるスタックの数は有限であることを前提にする．

この有限性の仮定はやや強い制約で，一般的にはなりたたないが，エージェントのアプリケーションは単純なループ構造をもつため，この制約内でもアプリケーションの記述には問題ないと考える．たとえばループを引き起こす*call*を全てサブプログラムの最後の行でしか許さないとするとこの制約は満たされる．

[3] メッセージベースの状態遷移関係
メッセージベースの状態の遷移はaddとrmのアクションによって引き起こされる．

エージェント*agt*のプログラムスタックの状態st_iには，現在実行中のif-thenルールが1つ対応する(スタックのtop)．$agt.st_i$ をエージェント*agt*の現在のプログラムスタックの状態がst_iであることを表すブール変数とする．

st_iで実行中のif-thenルールのテスト式をψとし，そのthen以下に*add(agt', m)*が現れていたとする．このとき，この*add*アクションによるメッセージベースの変化を， 記号⇒を使って，$agt.st_i\wedge bf(agt, \psi)\Rightarrow agt'.[m]^\wedge$という関係式で表す．else以下にaddアクションがある場合の議論は省略する．

もし，*rm*アクション*rm(m)* がthen以下に現れていたなら，このアクションによる変化を$agt.st_i\wedge bf(agt, \psi)\Rightarrow\neg agt'.[m]$で表す．else以下の場合の議論は省略する．

[4] 非同期性
エージェントのinterleavingモデルによる非同期性に関係して，各エージェント agt_k に対し，ブール変数run_kを導入する．run_kは agt_kが現在if-thenルールを実行中であることを表す．agt_kのプログラムから[2]，[3]で導かれた関係 $A\Rightarrow B$に対し，これを $run_k\wedge A\Rightarrow B$と書き換える．

[5] システム全体の遷移関係
エージェントagt_kに関する状態変数*x*に対して，

エージェントが動作していない場合の遷移関係として，$\neg run_k \wedge x \Rightarrow x^{\wedge}$を導入する．ここで，[4]で得られた関係と合わせて，その中で左の部分が同じものたち，即ち$A_1 \Rightarrow B, \ldots A_n \Rightarrow B$の形のものたちを，$B$ごとに分類し，その各々の分類に対し，論理式$B \text{->} A_1 \vee \ldots \vee A_n$を作る．$B$が負のリテラルの場合，少し工夫がいるがここではその議論は省略する．

また，interleaving modelに基づく非同期性の表現のために，論理式$\wedge_{i<j} \neg (run_i \wedge run_j)$および$\vee_i run_i$を導入する．

最後に，これまで導入した論理式全ての連言をとり，検証に必要な論理式を得る．

4. 状態遷移モデルの簡約化

簡約化について，そのアイデアだけを説明し，詳細は割愛する．

ここで導入した言語のプログラムは，変数を含んだ if-then ルールの形をしている．要求仕様は，一般にエージェントのメッセージベースについての時相的な状態（「エージェント *agt* はいつかはメッセージ *m* を受け取る」など）で表現される．このような要求仕様の中に含まれるメッセージを，指定されたエージェントをその受信者として，逆に送信者をたどることによって，この要求仕様に関係するプログラムの部分が求められる．ここで，仕様に関係するのは，送受信の連鎖の部分だけでなく，その送信アクションを行う if-then ルールの実行にかかわる *call* アクション，即ち，その if-then ルールを含むサブプログラムを *call* する連鎖，およびその if-then ルール実行前にループする *call* の遷移などである．

if-then ルールには変数が含まれるので，実際の手続きでは，consistent な unification を管理して，上記の連鎖の追跡が行われる．

5. プログラム言語 Erdoes と形式的検証に関する議論

３章で触れなかったが，実際の検証用の論理式の生成において，全てがプログラムから自動で行われるわけではなく，*ex_call* action で起動される外部メソッドの挙動に対しては，プログラマが，どのようなアウトプットがメッセージの形であり得るかを記述しなければならない．しかし，ここがこの言語のポイントで，エージェントが内部で行う単独の処理(外部メソッド)と，メッセージ交換によるエージェント間のインタラクション部分を分離し，検証の対象をインタラクションの部分にしぼることが基本のアイデアである．そして，エージェントが互いに疎結合であるという状況，即ち，特定のパターンのメッセージをもとに互いの行動を制御し，またメッセージベースも queue でなく，集合で十分であることを利用して，インタラクション部分の遷移の有限化がなされ，論理式生成の自動化が実現される（図１）．現在広く利用されているエージェント開発プラットフォームである Jade などを考えた場合，そのプログラムの形式的検証は現状のままでは不可能で，それを実現するためには，本論文で導入したプログラム言語の構造をなんらかの形で Jade に投影して，その中でプログラムを書くことになると考えている．

Erdoes は，Java 上のインタープリタの形で実装され，Jade と遜色の無いスピードで動いている（図２）．また，エージェントのモバイルや，ネームサーバの機能も備えており，さらにエージェントの国際標準である FIPA 仕様も満たし，Jade との相互接続が可能になっている．

図１：検証システム（BDD の表示）

図２：Erdoes エージェントの実行風景

参考文献

[1] T. Araragi, P.C. Attie, I. Keidar, K. Kogure, V. Luchangco, N.A. Lynch, and K. Mano, “On Formal Modeling of Agent Computations,” Formal Approaches to Agent-Based Systems: First International Workshop, FAABS, LNCS 1871, pp.48-62, 2000.

[2] K.L.McMillan, “Symbolic Model Checking,” Kluwer Academic Publishers, 1993.